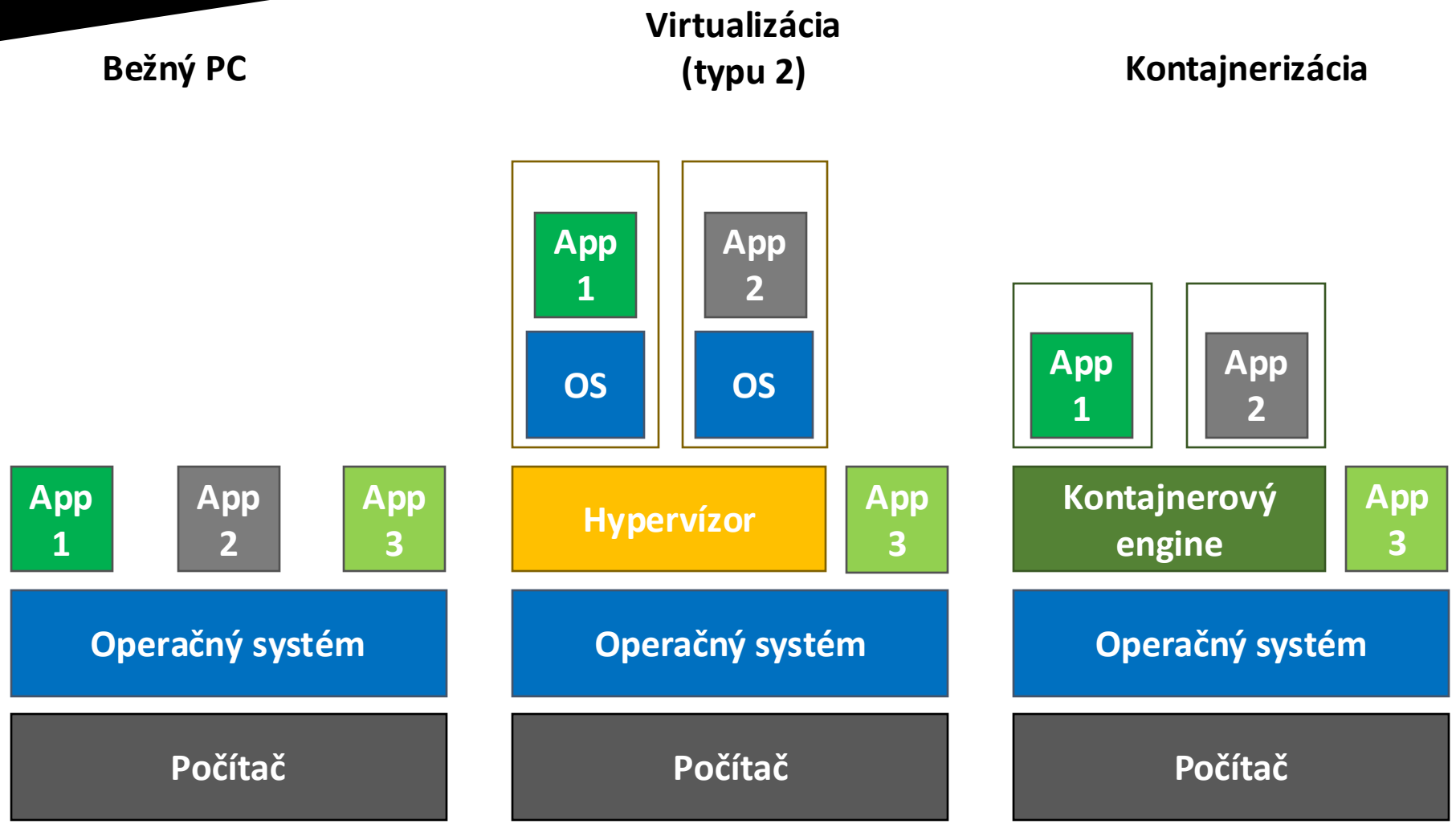
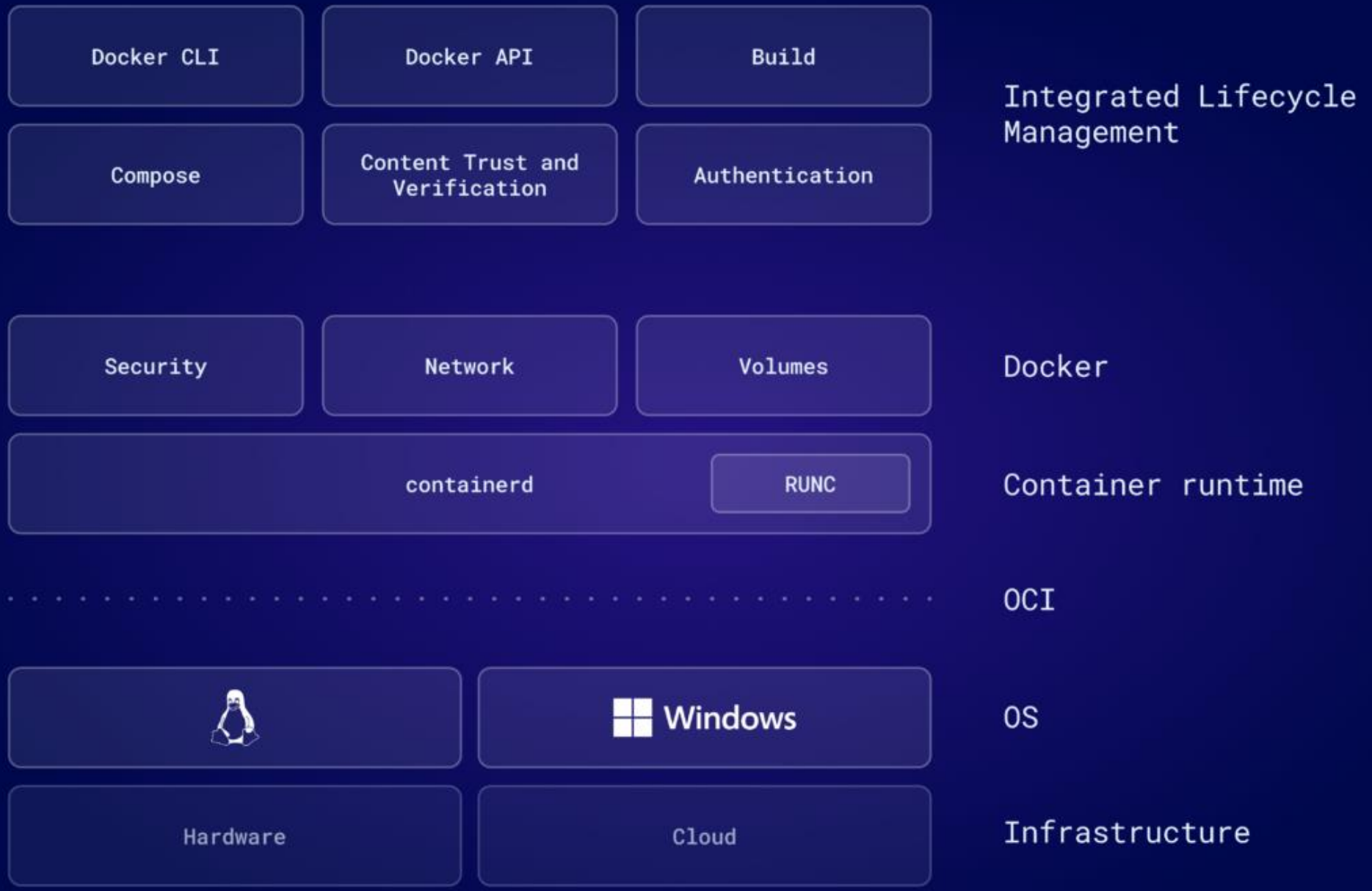
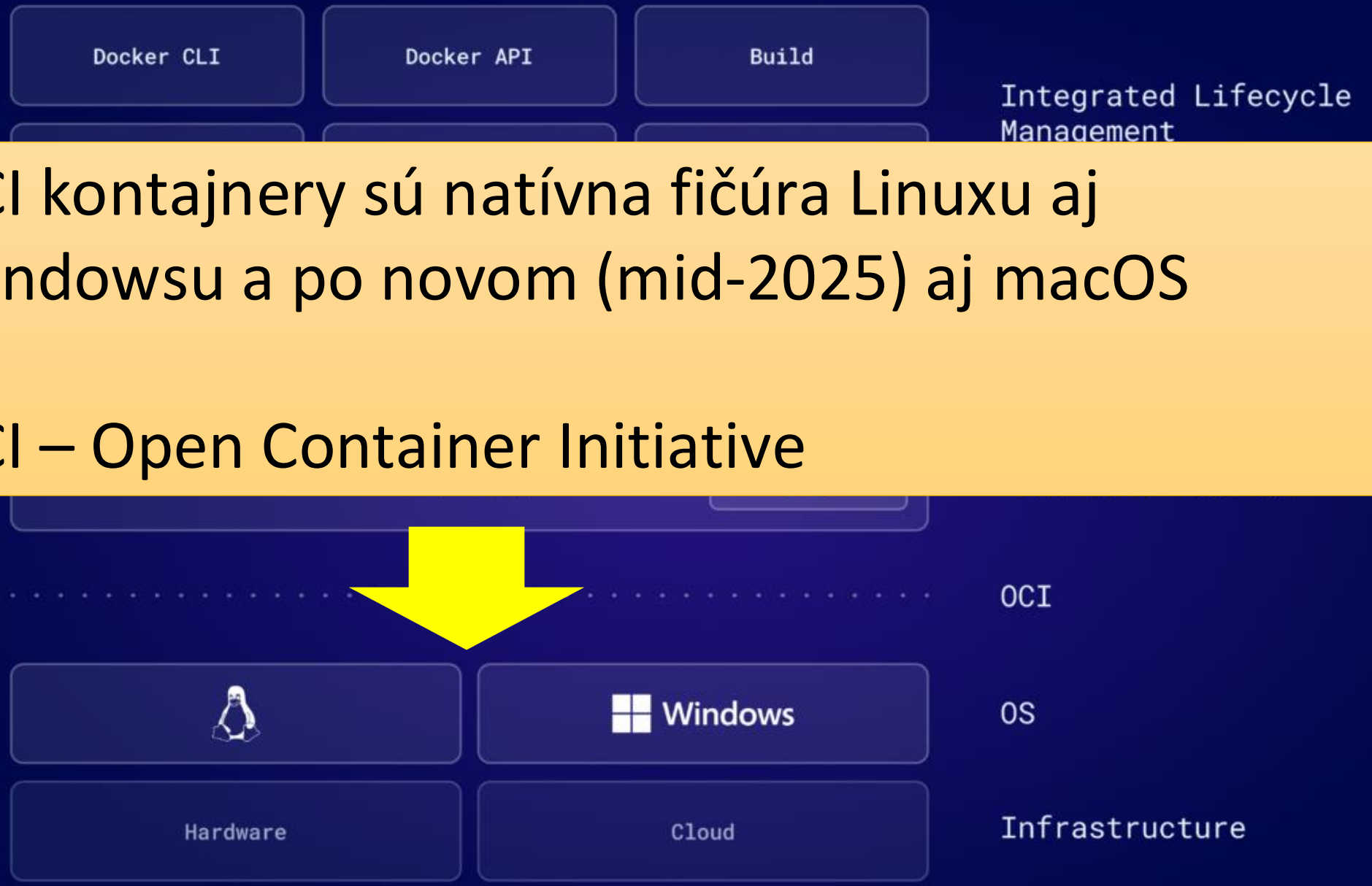
Kontajnerizujeme

Projekt 1

Kontajnerizácia




Docker CLI
Docker API
Build
Compose
Content Trust and Verification
Authentication
Integrated Lifecycle Management
Security
Network
Volumes
Docker
containerd
RUNC
Container runtime
OCI
Windows
OS
Hardware
Cloud
Infrastructure



OCI kontajnery sú natívna fičúra Linuxu aj Windowsu a po novom (mid-2025) aj macOS

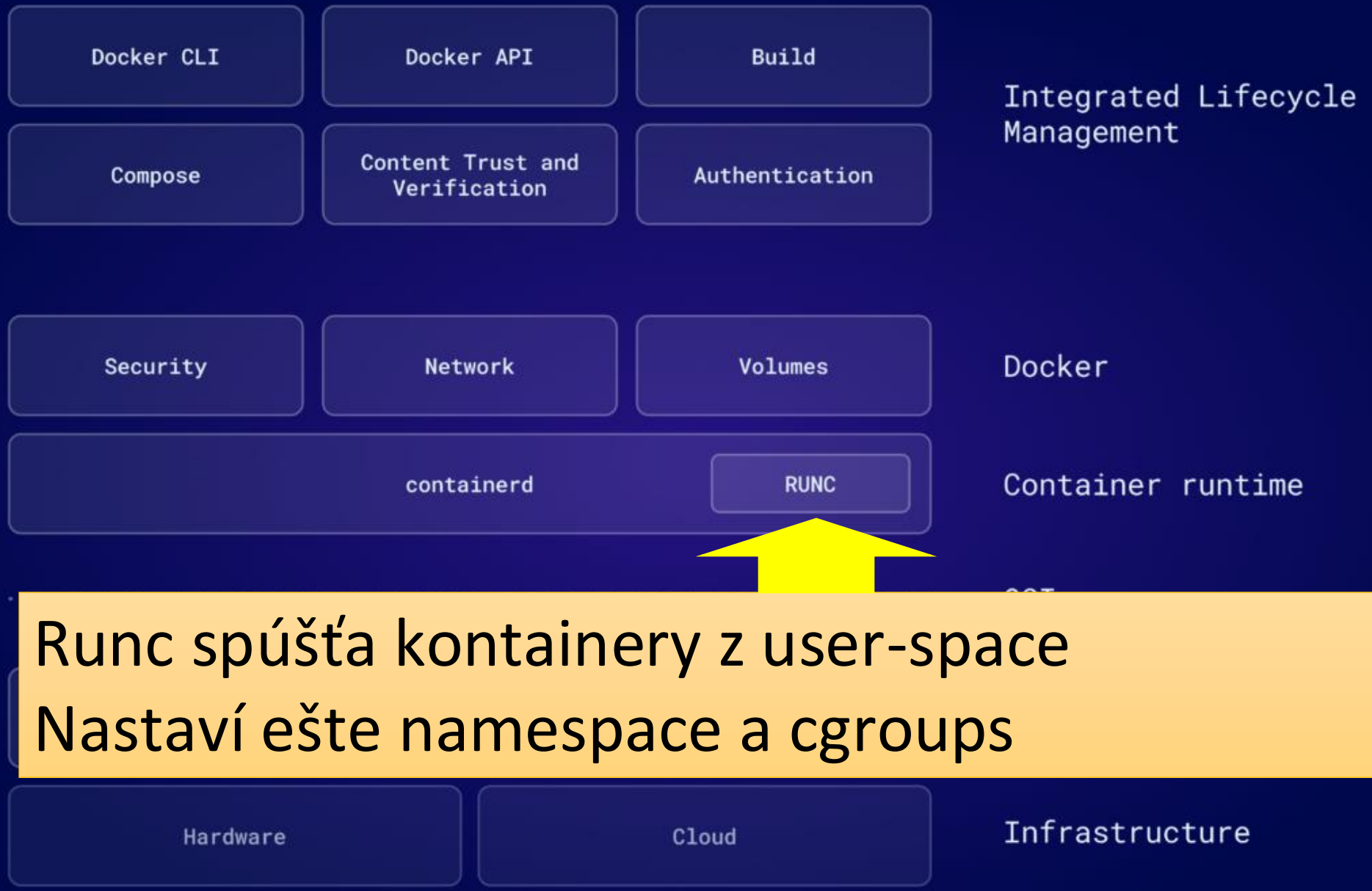
OCI – Open Container Initiative

Docker Engine

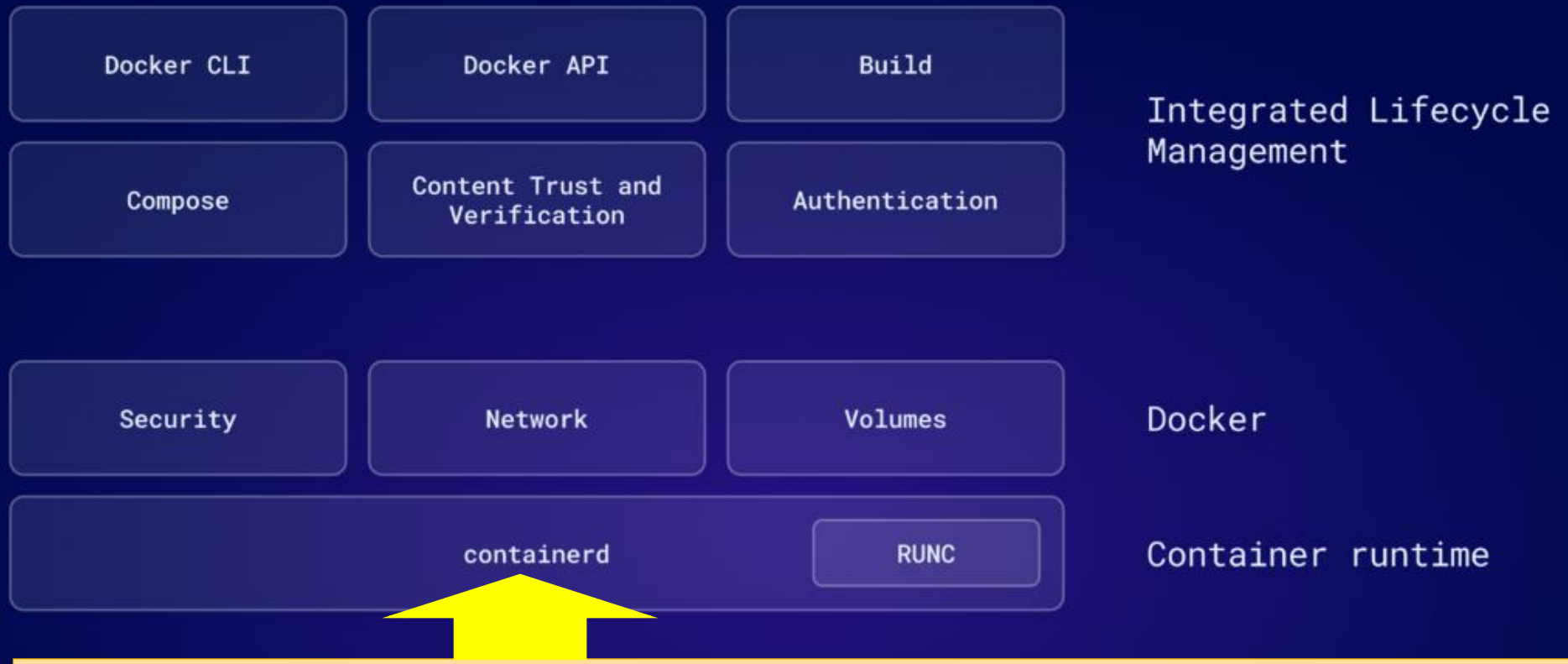
- CLI nástroj
- Apache 2.0 licencia
- Natívne na Linuxe
- Súčasťou distribučných repozitárov
- Docker Compose treba doinštalovať zvlášť
- Cross-platform (arm64, amd64) podpora cez QEMU

Docker Desktop

- GUI nadstavba
- Treba $$$ ak máte viac ako 250 zamestnancov
- Treba $$$ ak zarobíte viac ako $10 mil ročne
- Podporuje aj beh natívnych Windows kontajnerov
- DD natívne macOS kontajnery nepodporuje
  - Apple robí VM per kontajner
  - nemajú namespaces, cgroups
- Docker Desktop teda vždy spustí zdieľanú Linux VM
  - Na Linux pomocou KVM
  - Na Windows pomocou WSL2/HyperV
  - Na mac OS pomocou Virtualization.framework
- Cross-platform (arm64, amd64) podpora cez QEMU/Rosetta2



Runc spúšťa kontainery z user-space
Nastaví ešte namespace a cgroups



Containerd je démon, ktorý pracuje s obrazmi - images

- Obraz je definícia kontainera
  - Kontainer = bežiaci "mini OS" bez jadra
  - Obraz = z čoho "mini OS" máme vyskladaný

Základné Docker príkazy

- docker build - vytvorenie image z Dockerfile
- docker run - spustenie kontajnera z obrazu
- docker ps - zoznam bežiacich kontajnerov
- docker images - zoznam obrazov
- docker logs - výpis logov kontajnera
- docker stop / rm - zastavenie a zmazanie kontajnera

Docker Exec a Logs

- docker exec -it <container> sh/bash
  - Spustenie príkazu v bežiacom kontajneri
  - Cez (**-it**) a **bash/sh** otvoríme interaktívny shell
  - Používa sa na debugovanie
  - Nevytvára nový kontajner – pracuje s existujúcim

- docker logs -f <container>
  - Zobrazí logy z kontajnera a interaktívne ich následuje (**-f**)

Rootless Docker

- Rootless Docker – bez potreby root práv
  - Bezpečnejšie spúšťanie kontajnerov
  - Znižuje riziko privilege escalation
    - Ten kto vie spúšťať kontajner bez sudo (je v group docker), aj keď nie je sudo-er, si vie do neho mount-núť ľubovnoľný súbor/priečinok z hosta a editovať ho z kontajnera ako root
  - Nevyžaduje prístup k */var/run/docker.sock* ako root
  - Obmedzenia: niektoré sieťové a low-level funkcie nemusia fungovať

Podman – detailnejšie

- Podman je plne kompatibilný s OCI
- Daemonless – neexistuje centrálny docker daemon
  - Každý kontajner je bežný proces používateľa
- Podpora rootless režimu by default
- Kompatibilný s Docker CLI (alias docker=podman)
- Lepšia integrácia so systemd
- Podpora pod-ov (podobne ako Kubernetes)
  - Pod – skupina kontajnerov so zdieľaným úložiskom a localhostom
- Drobné otravné odlišnosti od Dockera

Základné obrazy

- Základný obraz – **scratch**
  - Prázdný userspace bez shellu, bez libc, nič...
- OS obrazy
  - Osekané Linuxové distribúcie - alpine, debian, ubuntu, ...
  - shell – sh (alpine), bash
  - package manager, základné Linux utilitky a knižnice

Aplikačné obrazy

- Typicky postavené nad OS obrazmi
- Pre nejakú konkrétnu appku, jazyk
  - mysql, redis, postgres, openjdk, python, node, ...
- Používame ich najčastejšie

Distroless obrazy

- Obraz konkrétnej appky odvodený priamo od **scratch**, alebo značne osekaným OS obrazom bez shellu, často aj bez libc, ...
- **Bezpečnosť - kde nie je shell, tam sa ťažko hekuje**
- Menej priestoru na disku, rýchlejšie sťahovanie/upload
- **Ťažko sa debugujú a riešia problémy**

Dockerfile – základná štruktúra

- FROM – základný image
  - RUN – vykonanie príkazu počas build
  - COPY / ADD – kopírovanie súborov
  - WORKDIR – nastavenie pracovného adresára
  - EXPOSE – otvorený port v kontajneri
  - CMD / ENTRYPOINT – čo sa spustí pri štarte

COPY vs ADD

- COPY – jednoduché kopírovanie súborov do image
  - COPY podporuje lokálne súbory a adresáre
  - ADD – robí to isté ako COPY + extra funkcionalita
  - ADD automaticky rozbalí .tar archívy
  - ADD umožňuje sťahovať súbory z URL (neodporúča sa)
  - Best practice: preferovať COPY, ADD len ak potrebujeme jeho extra vlastnosti

ENTRYPOINT vs CMD

- Čo sa má spustiť v obraze pri **docker run**

- CMD – default príkaz, ktorý sa spustí pri docker run
  - CMD sa dá prepísať parametrom pri docker run
- ENTRYPOINT – definuje hlavný proces kontajnera
  - ENTRYPOINT sa bežne nepoužíva na prepísanie, ale ako fixný vstupný bod
  - Kombinácia: ENTRYPOINT + CMD (CMD ako default argumenty)
  - Shell form vs exec form – odporúčaná je exec forma (['app','arg'])

Dockerfile – pre Spring Boot appku

Dockerfile-api-simple

```
FROM maven:3.9.12-eclipse-temurin-25

WORKDIR /app

RUN groupadd -r app && useradd -r -g app app

COPY pom.xml ./
RUN mvn -B -DskipTests dependency:go-offline

COPY src/ src/
COPY checkstyle-suppressions.xml checkstyle-suppressions.xml
RUN mvn -B -DskipTests package \
    && JAR_PATH="$(ls target/*.jar | grep -v 'original' | head -n 1)" \
    && cp "${JAR_PATH}" target/app.jar

USER app
EXPOSE 8080

ENTRYPOINT ["java", "-jar", "/app/target/app.jar"]
```

Vrstvy obrazov (Image Layers)

- Každý Docker image je zložený z vrstiev (layers)
  - Väčšina príkazov v Dockerfile vytvára novú vrstvu
    - RUN, COPY, ADD
  - Vrstvy sú cacheované – zrýchlenie build procesu
  - Copy-on-write – kontajner má vlastnú zapisovateľnú vrstvu
  - Zdieľanie vrstiev medzi obrazmi šetrí miesto
  - docker history <image> ukáže jednotlivé vrstvy
  - Zmena vrsty N invaliduje vrstvy N+1, N+2, ...

COPY . .

- Možeme urobiť v Dockerfile
  - 1 **COPY . .** # skopíruj všetko z lokálneho priečinka do obrazu
  - 2 **RUN mvn package**
- Pri každej zmene zdrojáka potrebuje Docker znovu stiahnuť všetky Maven knižnice

Cache optimalizácia

- Dockerfile (zjednodušené):
  - 1️⃣ FROM maven:3.9
  - 2️⃣ COPY pom.xml .
  - 3️⃣ RUN mvn dependency:go-offline
  - 4️⃣ COPY src/ .
  - 5️⃣ RUN mvn package

  - Zmeníme iba jeden súbor v src/:
  - → Vrstva 4 sa invaliduje
  - → Vrstva 5 sa rebuildne
  - → Vrstvy 1–3 ostávajú cacheované ✅

Nastavme ešte našu appku

- Najprv si ešte nastavme application.properties na DB

```
spring.application.name=entrance-pro1a

spring.datasource.url=${SPRING_DATASOURCE_URL:jdbc:mysql://localhost:3306/mydatabase}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME:myuser}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD:secret}
spring.jpa.hibernate.ddl-auto=update
```

Buildnime

$ docker build -t entrance-api:simple -f Dockerfile-api-simple .

Spustenie vlastného obrazu

- Spustime image
  - **docker** compose –f compose.yaml up
  - **docker** run \
    --network entrance-pro1a_default \
    -p 8080:8080 \
    -e SPRING_DATASOURCE_URL=jdbc:mysql://mysql:3306/mydatabase \
    entrance-api:simple

Docker sieť pre compose.yaml
docker network ls

Multistage Dockerfile

- Pozrime sa na image

$ docker image list | grep entrance-api:simple

entrance-api:simple ae79ada90e50 1.22GB 422MB U

Multistage Dockerfile

- Obraz je veľký lebo obsahuje veci nepotrebné pre beh appky
  - Maven, JDK, …
  - Stačí nám iba JRE

- Použitie viacerých FROM inštrukcií
  - Dočasný "builder" obraz s "tučným" JDK
  - Potom zkopírujeme JAR do "štíhleho" obrazu s JRE
- Bezpečnejší a čistejší výsledný obraz

1. stage - builder

Dockerfile-api

```
FROM maven:3.9.12-eclipse-temurin-25 AS builder

WORKDIR /app

RUN groupadd -r app && useradd -r -g app app

COPY pom.xml ./
RUN mvn -B -DskipTests dependency:go-offline

COPY src/ src/
COPY checkstyle-suppressions.xml checkstyle-suppressions.xml
RUN mvn -B -DskipTests package \
    && JAR_PATH="$(ls target/*.jar | grep -v 'original' | head -n 1)" \
    && cp "${JAR_PATH}" target/app.jar

USER app
EXPOSE 8080

ENTRYPOINT ["java", "-jar", "/app/target/app.jar"]
...
```

2. stage

Dockerfile-api

```
...

FROM eclipse-temurin:25-jre-alpine-3.23

WORKDIR /app

RUN addgroup -S app && adduser -S -G app app

COPY --from=builder /app/target/app.jar /app/app.jar

USER app
EXPOSE 8080

ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Buildnime

$ docker build -t entrance-api:multistage -f Dockerfile-api-multistage .

$ docker image list | grep entrance-api

entrance-api:multistage	23a5300eed0d	412MB	125MB
entrance-api:simple	ae79ada90e50	1.22GB	422MB

OCI Register

- Obrazy môžeme niekde ukladať
  - docker push IMAGE_NAME
- Open-source projekty často používajú Docker Hub - hub.docker.com
  - Treba sa registrovať
  - Potom musíme naše obrazy prefixovať názvom nášho konta
    - docker build -t KONTO/entrance-api:latest -f Dockerfile-api .
    - docker login -u KONTO
    - docker push KONTO/entrance-api:latest
- Alebo na ľubovoľný iný OCI register (Quay.io, JFrog, ...)

Docker a GitLab CI

- CI pipeline môže buildiť Docker images
  - docker build + docker push v CI
  - Použitie CI premenných pre tagovanie
  - Automatizované nasadzovanie
- GitLab Runner beží ako samotný kontajner, alebo ako Docker executor
  - Izolované CI joby
  - Jednoduchšia údržba a škálovanie
- Gitlab je aj OCI register sám o sebe
  - Treba v projekte/groupe povoliť prístup

Upravíme .gitlab-ci.yaml

```
...
build-image:
  stage: build
  script:
    - export  VERSION="$(date +%Y%m%d-%H%M%S)-${CI_COMMIT_SHORT_SGA}"

    - echo "$CI_REGISTRY_PASSWORD" | docker login "$CI_REGISTRY" -u "$CI_REGISTRY_USER" --password-stdin

    - docker build -f Dockerfile-api-ci -t "$CI_REGISTRY_IMAGE/entrance-api:$VERSION" .

    - docker push "$CI_REGISTRY_IMAGE/entrance-api:$VERSION"
```

Nejdze to!!!!!!!!!

Docker in Docker (DinD)

- Gitlab runner spúšťa joby v dockri
- Potrebujeme aby náš image mal Docker
- Navyše Docker je démon
  - Potrebujeme špeciálny obraz, ktorý poskytuje Docker-in-Docker démona

- Spustenie Docker démona v kontajneri
  - Používa sa v CI prostredí
  - Alternatíva: socket binding (/var/run/docker.sock)
    - Dáme jobom prístup k Docker démonovi na hostovi
    - Elegantnejšie no nebezpečnejšie - nerobte to

```yaml
variables:
  ...
  DOCKER_HOST: tcp://docker:2375
  DOCKER_TLS_CERTDIR: ""

build-image:
  stage: build
  image:
    name: docker:29.2.1
    pull_policy: if-not-present
  services:
    - name: docker:29.2.1-dind
      pull_policy: if-not-present
      alias: docker
  script:
    ...
```

"docker" príkazy budú používať DIND démona

V tomto jobe nám netreba Javu, stačí nám Docker

DIND démon bežiaci v Dockri, kde je spustený job

Docker in Docker (DinD) vs Socket Binding – bezpečnosť

- DinD – Docker daemon beží vo vnútri kontajnera
  - Vyžaduje privileged mode → veľké bezpečnostné riziko
  - Privileged kontajner má takmer plný prístup k hostovi
- Socket binding – mount /var/run/docker.sock do kontajnera
  - Kontajner získa plnú kontrolu nad Docker daemonom hosta
  - Ak je kontajner kompromitovaný → útočník môže ovládnuť host
- Bezpečnejšia alternatíva:
  - buildkit, buildah – na vytváranie obrazov, nepodpora Testcontainers
  - rootless Docker, Podman – celkom fajn, nejaké edge cases

TestContainers v CI

- Máme test-java job
- Pre ten potrebujeme obraz s Java aj Dockerom kvôli Testcontainers
  - Obraz **docker:29.2.1** má Docker, nie Java
  - Obraz **maven:3.9.12-eclipse-temurin-25** má Java, nie Docker

- Buď si nájdeme obraz s obomi.
- Alebo si ho vyrobme
  - Lepšie, lebo potom budeme potrebovať aj Node

Builder obraz v projekte

```
FROM debian:trixie

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update \
&& apt-get install -y --no-install-recommends \
ca-certificates \
curl \
git \
gnupg \
&& rm -rf /var/lib/apt/lists/*

RUN apt-get update \
&& apt-get install -y --no-install-recommends \
openjdk-25-jdk \
checkstyle \
maven \
&& rm -rf /var/lib/apt/lists/*

# Install Docker CLI pinned to 29.2.1 from Docker's official APT repo.
RUN install -m 0755 -d /etc/apt/keyrings \
&& curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc \
&& chmod a+r /etc/apt/keyrings/docker.asc \
&& echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/debian trixie stable" > /etc/apt/sources.list.d/docker.list \
&& apt-get update \
&& apt-get install -y --no-install-recommends docker-ce-cli=5:29.2.1-1~debian.13~trixie \
&& rm -rf /var/lib/apt/lists/*

# Defaults for GitLab docker:dind service integration.
ENV DOCKER_HOST=tcp://docker:2375
ENV DOCKER_TLS_CERTDIR=""
```

Job v Gitlab CI

```
builder:
  stage: builder
  image:
    name: docker:29.2.1
    pull_policy: if-not-present
  services:
    - name: docker:29.2.1-dind
      pull_policy: if-not-present
      alias: docker
  rules:
    - changes:
        - Dockerfile-builder
    - when: manual
  script:
    - export VERSION="$(date +%Y%m%d-%H%M%S)-${CI_COMMIT_SHORT_SHA}"
    - echo "$CI_REGISTRY_PASSWORD" | docker login "$CI_REGISTRY" -u "$CI_REGISTRY_USER" --password-stdin
    - docker build --provenance=false --sbom=false -f Dockerfile-builder -t "$CI_REGISTRY_IMAGE/builder:$VERSION" .
    - docker push "$CI_REGISTRY_IMAGE/builder:$VERSION"
```

Použitie buildera pre ostatné joby

```
default:
  image:
    name: gitlab.science.upjs.sk:4567/pro1a-2026/entrance-pro1a/entrance-api:20260301-170114-b5679cf1
    pull_policy: if_not_present
  tags:
    - docker
  services:
    - name: docker:29.2.1-dind
      pull_policy: if-not-present

...
```

(Voliteľné) Prepoužitie artefaktu z jobu

- V jobe **build-api** urobíme artefakt
  - Ale **build-image** to buildí znova
- S nejakou verziou z **.before_script**
  - Ale každý job si urobí vlastnú premennú VERSION

```
build-api:
  stage: build
  script:
    - export VERSION="$(date +%Y%m%d-%H%M%S)-${CI_COMMIT_SHORT_SHA}"
    ...
    - mv target/entrance-pro1a-${VERSION}.jar target/app.jar
  artifacts:
    reports:
      dotenv: build.env
    paths:
      - target/app.jar
```

(Voliteľné) Prepoužitie artefaktu z jobu

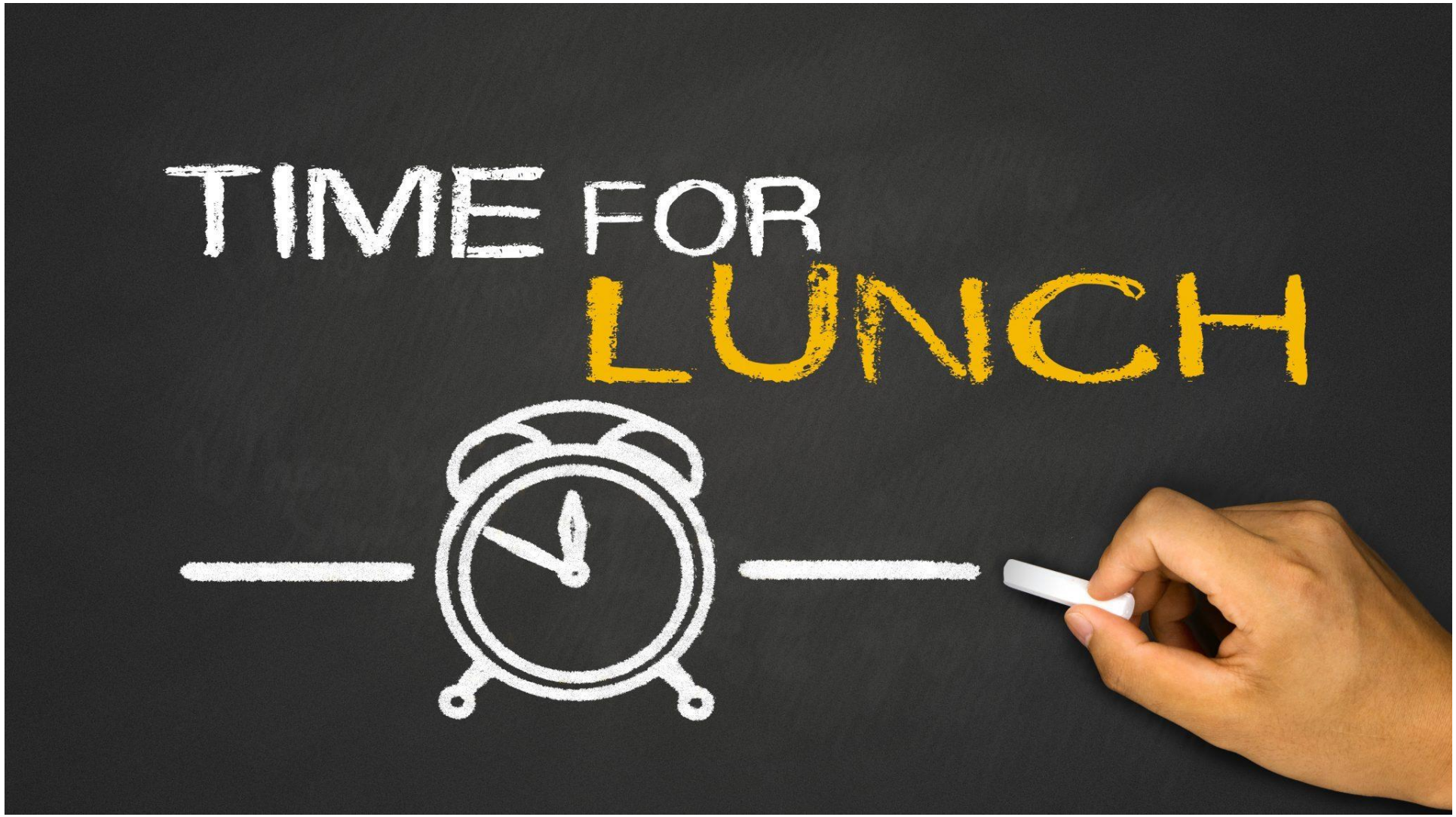
```yaml
build-image:
  stage: build
  needs:
    - job: build-api
      artifacts: true
  script:
    - echo "$CI_REGISTRY_PASSWORD" | docker login "$CI_REGISTRY" -u "$CI_REGISTRY_USER" --password-stdin
    - docker build -f Dockerfile-api-ci -t "$CI_REGISTRY_IMAGE/entrance-api:$VERSION" .
    - docker push "$CI_REGISTRY_IMAGE/entrance-api:$VERSION"
```

Dockerfile-api-ci

```
FROM eclipse-temurin:25-jre-alpine-3.23
WORKDIR /app
RUN addgroup -S app && adduser -S app -G app
COPY target/app.jar /app/app.jar
USER app
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

🎓 Otázky

- 1️⃣ Vysvetlite prečo je poradie COPY a RUN dôležité pre cache.
- 2️⃣ Aký je rozdiel medzi ENTRYPOINT a CMD?
- 3️⃣ Prečo je Docker in Docker v privileged režime rizikový?
- 4️⃣ Kedy by ste použili multistage build?


TIME FOR
LUNCH